Stack

Uno stack (pila) è un insieme dinamico in cui è possibile accedere solamente all'ultimo elemento inserito.
Di conseguenza gli elementi possono essere tolti solo con modalità LIFO (*Last In First Out*).

Le operazioni ammesse su uno stack sono:

Operazione	Significato
`newStack()`	crea uno stack vuoto
`stackPush(el)`	inserisce l'elemento *el* in cima allo stack
`stackTop()`	restituisce il valore dell'elemento in cima allo stack
`stackPop()`	toglie l'elemento in cima allo stack
`stackIsEmpty()`	restituisce 1 se lo stack è vuoto, 0 altrimenti
`stackDestroy()`	elimina lo stack

Il modulo stack.c

Le operazioni della tabella possono essere implementate in un modulo (file) `stack.c`.

Deve essere definito lo header file `stack.h` contenenete le definizioni e i prototipi necessari per utilizzare lo stack.

Uno stack è descritto dal tipo `stack` da definire in `stack.h`.

> La definizione del tipo `stack` dipende da come lo stack è implementato e può essere ignorata dall'utente.

In C non è possibile descrivere uno "stack generico", utilizzabile indipendentemente dal *tipo* degli elementi.

Per semplicità, consideriamo il caso in cui gli elementi dello stack sono interi.

Funzioni disponibili su uno stack di `int`

- `stack *newStack()`
  Crea uno stack vuoto e restituisce il suo indirizzo.
- `void stackPush(int n, stack *s)`
  Inserisce l'elemento `n` nello stack `s`.
  Il valore di `s` non è modificato.
- `int stackTop(stack *s)`
  Restituisce il valore dell'elemento in cima allo stack `s`.
- `void stackPop(stack *s)`
  Toglie l'elemento in cima allo stack `s`.
  Il valore di `s` non è modificato.
- `int stackIsEmpty(stack *s)`
  Restituisce 1 se lo stack `s` è vuoto, 0 altrimenti.
- `void stackDestroy(stack *s)`
  Elimina lo stack `s`.

Esempio 1: inversione di una lista

Scrivere una funzione

```
void printInv(element *h)
```

che stampa la lista (semplice) di interi `h` in ordine inverso.

Soluzione 1: uso della ricorsione

È la soluzione più semplice.

Si usa uno schema ricorsivo già visto in altri esempi:
- se la lista `h` è vuota non si fa nulla;
- altrimenti si stampa in ordine inverso con una chiamata ricorsiva la lista `h->next` (lista che parte dal secondo elemento di `h`) e poi si stampa `h->info` (primo elemento elemento della lista `h`).

```
void printInv(element *h){
  if(h !=  NULL ){
    printInv(h->next);
    printf("%d ", h->info);
  }
}
```

Soluzione 2: utilizzo di uno stack

Se non si vuole utilizzare la ricorsione, occorre utilizzare delle strutture ausiliarie per memorizzare gli elementi della lista.

La struttura più adatta allo scopo è uno stack St di interi.

Infatti:

(1) Si crea uno stack St vuoto.

(2) Si attraversa la lista dall'inizio alla fine e si inseriscono gli elementi visitati in St.

(3) Fintanto che St non è vuoto, si estrae un elemento n da St e si stampa n.

È facile verificare che gli elementi sono stampati in ordine inverso rispetto alla loro posizione nella lista.

Il codice della funzione (*indipendentemente* da come sia realizzato lo stack di interi) è:

```
void printInv(element *h){
  stack *st; // indirizzo dello stack
  st = newStack(); // crea lo stack vuoto

 while(h != NULL){
    stackPush(h->info, st);
    h = h->next;
  }

while(!stackIsEmpty(st)){
    printf("%d ", stackTop(st));
    stackPop(st);
  }
 stackDestroy(st); // lo stack non serve piu'
}
```


Esempio 2: visita preorder di un albero

Per definire un algoritmo per la visita in ordine anticipato (preorder) di un albero binario senza usare la ricorsione, si può utilizzare uno stack per gestire l'ordine con cui i nodi devono essere visitati.

Nello stack memorizziamo gli *indirizzi* dei nodi da visitare (e non i valori dei nodi).

Occorre quindi modificare il codice che gestisce lo stack facendo in modo che gli elementi abbiano tipo `node*` anziché `int`.

Ad esempio, il primo parametro della funzione `stackPush()` deve avere tipo `node*`.

Descrizione dell'algoritmo

Supponiamo di voler visitare in preorder un albero non vuoto di radice `r`.

(1) Si inserisce nello stack la radice `r`.

(2) Finché lo stack non è vuoto:

(2.1) Si toglie dallo stack un nodo.

(2.2) Sia `p` il nodo estratto al punto precedente.

Si visita `p` e si inseriscono nello stack il figlio destro di `p` (se definito) e il figlio sinistro di `p` (se definito).

Esercizio

Verificare che la procedura è corretta provando a simulare l'algoritmo con degli esempi.
Cosa succede se in (2.2) si inverte l'ordine con cui i figli di `p` sono inseriti nello stack?

La funzione preorderIt() visita in ordine anticipato un albero binario non vuoto di radice r eseguendo su ogni nodo una funzione visita() passata come secondo parametro.

```
void preorderIt(node *r , void visita(node *)) {
    node *p;
    stack *st;
    st = newStack(); // crea lo stack vuoto
    stackPush(r, st);
    while(!stackIsEmpty(st)) {
            p = stackTop(st);
            stackPop(st);
            visita(p);
            if (p->right != NULL)
               stackPush(p->right, st);
            if (p->left != NULL)
               stackPush(p->left, st);
        }
    stackDestroy(st);  // lo stack non serve piu'
}
```


Implementazione di uno stack di interi: array

Gli elementi dello stack sono memorizzati in un array allocato dinamicamente di dimensione STACKSIZE (parametro definito nel file stack.c).

Il tipo stack è definito da una struttura struct stack avente i campi vett e top.

- Il campo vett contiene l'indirizzo dell'array contenente i dati. Poiché l'array contiene interi, vett deve avere tipo int*.
- Il campo top contiene l'indice del primo elemento libero dell'array vett. Il valore di top va da 0 a STACKSIZE-1; l'indice dell'elemento in cima allo stack è top-1.

```
struct stack {
  int top;
  int *vett;
};

typedef struct stack stack;
```


Creazione stack vuoto

Lo stack vuoto è rappresentato da un elemento di tipo stack creato dinamicamente in cui top vale 0 e vett contiene l'indirizzo di un vettore di STACKSIZE elementi allocato dinamicamente.

La funzione newStack() restituisce l'indirizzo di uno stack vuoto.

```
stack  *newStack(){
  stack  *s;
  s = malloc(sizeof(stack));
  s->top = 0;
  s->vett = calloc(STACKSIZE, sizeof(int));
  return s;
}
```


Stato dello stack

La funzione stackIsEmpty() restituisce 1 se lo stack s è vuoto, 0 altrimenti.

```
int stackIsEmpty(stack *s){
  return s->top == 0;
}
```

È utile definire anche la funzione stackIsFull()che restituisce 1 se lo stack s è pieno, 0 altrimenti.

```
int stackIsFull(stack *s){
  return s->top == STACKSIZE;
}
```

Le funzioni per verificare lo stato di uno stack servono per evitare situazioni inconsistenti che possono causare errori in fase di esecuzione (ad esempio, chiamata di stackTop() o stackPop() su uno stack vuoto).

Push

La funzione `stackPush()` inserisce l'elemento `n` nello stack `s`. Se lo stack è già pieno si verifica una situazione di errore.
Il valore di `s` non cambia.

```
 void stackPush(int n, stack *s){
   if(stackIsFull(s)) { // stack pieno
     printf("Stack pieno\n");
     exit(-1); // il programma termina restituendo -1
   }
   else{  // poni n in cima a s
     s->vett[s->top] = n;
     (s->top)++; // prossima posizione libera
   }
}
```

La gestione della situazone di errore può essere migliorata.

Top

La funzione `stackTop()` restituisce il valore dell'elemento in cima allo stack `s`.

```
int stackTop(stack *s){
  return s->vett[(s->top)-1];
}
```

Pop

La funzione `stackPop()` toglie l'elemento in cima allo stack `s`. Il valore di `s` non cambia.

```
void stackPop(stack *s){
  (s->top)--;   // prossima posizione libera
}
```

Eliminazione dello stack

Occorre eliminare prima il vettore `s->vett`, quindi la struttura `s`.

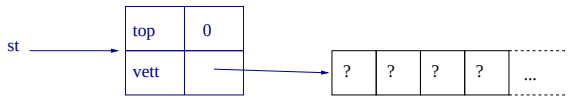
```
void stackDestroy(stack *s){
  free(s->vett);
  free(s);
}
```


Esempio

Dopo le istruzioni

```
stack *st;
st = newStack();
```

in memoria si ha

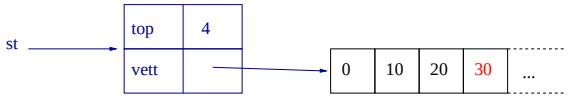


che rappresenta uno stack vuoto (i valori dell'array `st->vett` sono indefiniti).

Dopo le istruzioni

```
stackPush(0,st);
stackPush(10,st);
stackPush(20,st);
stackPush(30,st);
```

si ha



Infatti, sono stati inseriti nello stack 4 elementi (gli interi 0, 10, 20 e 30).

La prossima posizione libera è la locazione `st->vett[4]` mentre l'elemento in cima allo stack è quello in `st->vett[3]`.

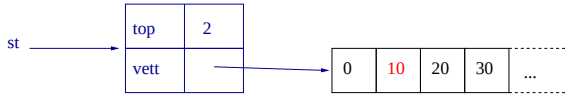
Nota

`st->vett[4]` equivale a `(st->vett)[4]` (vedere la tabella degli operatori).

Eseguendo

```
stackPop(st);
stackPop(st);
```

si ha:

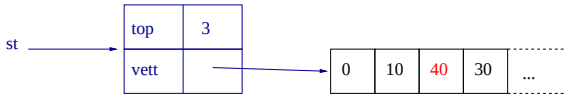


Essendo il valore di `st->top` uguale a 2, l'elemento in cima allo stack è 10 (elemento in `st->vett[1]`), la prossima locazione libera è `st->vett[2]`.

Eseguendo ora

```
stackPush(40, st);
```

in memoria si ha



L'elemento in cima allo stack è 40 (elemento in `st->vett[2]`), la prossima locazione libera è `st->vett[3]`.

Modifica della dimensione di un vettore allocato dinamicamente

L'obiettivo è quello di gestire il caso in cui viene richiesta l'operazione `stackPush()` su uno stack pieno, evitando l'interruzione del programma.

Una possibile strategia è quella di "aggiungere" `STACKSIZE` nuovi elementi all'array che implementa lo stack ogni volta che si verifica tale situazione.

Occorre modificare la struttura `stack` in modo che memorizzi l'attuale dimensione dello stack.

```
struct stack {
  int top;
  int *vett;
  int dim;  // dimensione attuale dello stack
};
```

Questo comporta anche la modifica del codice di `stackIsFull()`.

Per "ingrandire" un array `oldVett` allocato dinamicamente, si può definire una funzione

```
int *vettResize(int oldVett[], int oldSize, int newSize)
```

che, dato un array di interi `oldVett` avente `oldSize` elementi, crea dinamicamente un array di interi `newVett` avente `newSize` elementi, con `oldSize<newSize`, tale che:

```
newVett[0] == oldVett[0],
newVett[1] == oldVett[1],
          ...
newVett[oldSize-1] == oldVett[oldSize-1].
```

L'array `oldVett` viene quindi cancellato e la funzione restituisce l'indirizzo di `newVett` oppure `NULL` se non è stato possibile creare il nuovo array.

Il codice di `stackPush()` va modificato come segue:

```
 void stackPush(int n, stack *s){
   if(stackIsFull(s)) { // stack pieno
      s->vett = vettResize(s->vett, s->dim, (s->dim) + STACKSIZE);
      if(s->vett == NULL){// errore creazione nuovo vettore
            ... MESSAGGIO ERRORE ...
            exit(-1);
          }
      else // aggiorna dimensione dello stack
       s->dim =  (s->dim) + STACKSIZE;
   }
  ...
}
```


La funzione `realloc()`

Per modificare la dimensione di un array allocato dinamicamente si può usare la funzione standard `realloc()`. Vedere la documentazione sui manuali.

In `stackPush()`, per aumentare la dimensione del vettore `s->vett` si può fare:

```
s->vett = realloc(s->vett, ((s->dim) + STACKSIZE) * sizeof(int) );
```

Non è necessario specificare la dimensione del vettore da ridimensionare.

Nota

La funzione `realloc()` può essere onerosa, in quanto può essere necessario allocare una nuova area di memoria e copiare in essa i vecchi valori del vettore.

Pertanto `realloc()` va usata con cautela e la nuova dimensione del vettore deve essere *molto più grande* dell'attuale, per evitare di dover eseguire in tempi brevi una nuova riallocazione.

Stack implementato mediante lista

Gli elementi dello stack sono rappresentati mediante una lista di interi.

La struttura `stackElement` descrive un elemento della lista.

```
struct stackElement {
    int info;
    struct stackElement *next;
};

typedef struct  stackElement   stackElement;
```

Per definire lo stack usiamo la struttura

```
struct stack {
  struct stackElement *top; // primo elemento della lista
};

typedef struct stack  stack;
```


Creazione stack vuoto

La funzione `newStack()` restituisce l'indirizzo di uno lo stack vuoto.

```
stack  *newStack(){
  stack *s;
  s = malloc(sizeof(stack));
  s->top = NULL; // lista vuota
  return s;
}
```

Push

La funzione `stackPush()` inserisce l'elemento `n` in cima allo stack `s`. Il valore di `s` non cambia.
Il nuovo elemento viene inserito in testa alla lista `s->top`.

```
void stackPush(int n, stack *s){
  stackElement *new = malloc(sizeof(stackElement));
  new->info = n;
  new->next = s->top;
  s->top = new;
}
```

Top

La funzione `stackTop()` restituisce il valore dell'elemento in cima allo stack `s` (che è il primo elemento della lista `s->top`).

Si assume che lo stack non sia vuoto (se `s->top` vale `NULL`, l'accesso a `s->top->info` dà errore in esecuzione).

```
int stackTop(stack *s){
    return s->top->info;
  }
```

Pop

La funzione `stackPop()` toglie l'elemento in cima allo stack `s` (il primo elemento della lista `s->top`). Il valore di `s` non cambia.

Si assume che lo stack non sia vuoto.

```
void stackPop(stack *s){
    stackElement *x;
    x = s->top; // elemento  da togliere
    s->top = s->top->next;
    free(x);
  }
```


Verifica stack vuoto

La funzione `stackIsEmpty()` restituisce 1 se lo stack `s` è vuoto, 0 altrimenti.

```
int stackIsEmpty(stack *s){
    return s->top == NULL;
}
```

Eliminazione dello stack

La funzione `stackDestroy()` cancella lo stack `s`.

```
void stackDestroy(stack *s){
  stackElement *x, *p;

// cancella tutti gli elementi di s->top
  x = s->top;
  while(x != NULL){
      p = x->next;
      free(x);
      x = p;
    }
  free(s); // cancellalo stack
}
```


Confronto fra le due soluzioni

Entrambe le soluzioni sono efficienti in quanto tutte le operazioni avvengono in *tempo costante*.

- L'uso della lista permette una migliore gestione dello spazio. Infatti, per rappresentare uno stack contenente n elementi, lo spazio richiesto è $\Theta(n)$.

  Con l'array invece si può avere spreco di memoria.

- La versione con l'array è però più efficiente in termini di tempo.

  Infatti, usando le liste le operazioni di `stackPush()` e `stackPop()` richiedono rispettivamente l'esecuzione di una `malloc()` e una `free()` che hanno un certo costo (infatti, deve essere aggiornato dinamicamente lo stato dello heap del sistema).

  Con l'array dinamico invece l'allocazione di memoria viene fatta quando viene creato lo stack vuoto oppure quando lo stack deve essere ingrandito.

Esercizi

1. Scrivere un programma `inverti.c` che crea una lista di 2^n elementi contenente gli interi da 1 a 2^n, dove n è letto da standard input, stampa la lista in ordine inverso usando tre versioni della funzione `printInv()` (ricorsiva, con stack realizzato con un array, con stack realizzato con una lista) e calcola i rispettivi tempi di esecuzione. Nella versione in cui lo stack è implementato mediante un array, la dimensione dello stack è definita da `STACKSIZE`.

2. Eliminare da `inverti.c` le istruzioni di stampa ed eseguire il programma con liste di grosse dimensioni (esempio, liste con 2^{15} elementi). Confrontare i tempi di esecuzione e motivare i risultati ottenuti

Nota

Con liste molto grosse, la versione ricorsiva fallisce in quanto viene esaurita la memoria disponibile nello stack di sistema (questo perché sono attive contemporaneamente troppe chiamate ricorsive). La versione che risulta essere più efficiente è quella che usa lo stack implementato con array; spiegarne il motivo.

3. Modificare il programma `inverti.c` facendo in modo che l'array che implementa lo stack aumenti di dimensione quando viene richiesto l'inserimento di un elemento in uno stack pieno.

4. Scrivere un programma analogo a `inverti.c` per valutare le prestazioni di tre diverse funzioni per la visita in ordine anticipato di un albero binario.

5. Lo scopo dell'esercizio è quello di scrivere un programma `espressioni.c` per calcolare espressioni aritmetiche in notazione *postfissa* (notazione polacca), ossia in cui l'operatore è scritto dopo i suoi argomenti.

Come operatori consideriamo gli operatori binari per le operazioni fondamentali fra interi:

`+ (somma), - (differenza), * (prodotto), / (divisione), % (modulo)`

Il programma legge da standard input un'espressione in forma postfissa, calcola il suo valore e lo stampa. I simboli in input devono essere separati da uno o più caratteri di spaziatura (spazio, a capo, ecc.) e l'input deve terminare con il carattere `f`.

Esempi di espressioni di input:

```
2 3 + f   1 3 4 - + f  7 2 + 5 * f   2 1 - 4 * 10 7 8 + / + f
```

Le espressioni sono rispettivamente equivalenti a

```
2+3       1+(3-4)      (7+2)*5       ((2-1)*4) + (10/(7+8))
```


Per calcolare il valore di un'espressione si può eseguire il seguente ciclo che richiede l'uso di uno stack di interi St.

- Leggi il prossimo simbolo di input s.

  (1) Se s è un intero, inserisci s in St.
  (2) Se s è un'operazione `Op`:
    (2.1) Togli da St due interi a e b.
    (2.2) Calcola

    ```
    b Op a
    ```

    e inserisci il risultato in St.

Il ciclo di lettura termina quando viene letto il carattere `f`.

Al termine della lettura dell'espressione in input, St deve contenere un solo valore, che è il valore dell'espressione.

Nota

Se in (2.1) non è possibile togliere due elementi da St oppure se al termine della lettura dell'espressione St contiene più di un valore, l'espressione in input non è corretta.

Un simbolo s di input può essere rappresentato nel programma da un intero secondo la seguente codifica:

$$Code(s) = \begin{cases} 1 & \text{se } s \text{ è il carattere } + \\ 2 & \text{se } s \text{ è il carattere } - \\ 3 & \text{se } s \text{ è il carattere } * \\ 4 & \text{se } s \text{ è il carattere } / \\ 5 & \text{se } s \text{ è il carattere } \% \\ 6 & \text{se } s \text{ è il carattere } \texttt{f} \\ n+6 & \text{se } s \text{ rappresenta il numero } n \text{ e } n \geq 1 \\ n & \text{se } s \text{ rappresenta il numero } n \text{ e } n \leq 0 \end{cases}$$

In questo modo ogni simbolo di input è codificato in modo univoco da un intero. Per maggior leggibilità del codice, conviene introdurre le definizioni:

```
#define PIU 1
#define MENO 2
....
#define END 6
```

a. Verificare con degli esempi la correttezza dell'algoritmo proposto.

b. Scrivere il codice della funzione

```
int nextSymbol()
```

che legge da standard input il prossimo simbolo e restituisce l'intero che lo codifica.

Suggerimento. Conviene tentare la lettura di un intero con

```
scanf("%d", &n)
```

Se la lettura ha successo, è stato letto un intero e va restituita la codifica di *n* (ad esempio, se *n* vale 25 deve essere restituito 31). Altrimenti, il simbolo in input è uno dei simboli speciali; va quindi letto con `getchar()` e, a seconda del simbolo letto, va restituito `PIU` oppure `MENO` ecc.

c. Scrivere il codice della funzione

```
void  evaluateSymbol(int n, stack *st)
```

che, dato un intero `n`, compie una delle operazioni (1) o (2) scritte prima usando lo stack di interi `st`, in base al significato del simbolo *s* codificato da `n`.

Se i punti precedenti sono stati svolti correttamente, il programma `espressioni.c` ha la struttura:

```
...

int main(){
 stack* stack;
 int n;
 stack = newStack(); // crea stack vuoto
 while((n=nextSymbol()) !=  END) // lettura prossimo simbolo
    evaluateSymbol(n, stack)  //  esegui (1) oppure (2)
 printf("%d", stackTop(stack));  // stampa risultato
 stackDestroy(stack);
 return 0;
}
```

Per una soluzione alternativa, vedere paragrafo 10.6 del libro di testo (*"C - Didattica e Programmazione"*).

d. Migliorare il programma `espressioni.c` facendo in modo che vengano rilevate le situazioni di errore evidenziate nella nota precedente.

Coda

Una coda è un insieme dinamico in cui si può accedere solamente all'elemento che da più tempo è nell'insieme, quindi gli elementi possono essere tolti solo con modalità FIFO (*First In First Out*).

Le operazioni ammesse sono:

Operazione	Significato
`newQueue()`	crea una coda vuota
`queueIn(el)`	inserisce l'elemento *el* nella coda
`queueFirst()`	restituisce il valore del primo elemento della coda
`queueOut()`	toglie dalla coda il primo elemento
`queueIsEmpty()`	restituisce 1 se la coda è vuota, 0 altrimenti
`queueDestroy()`	elimina la coda

Funzioni disponibili su una coda di int

Una coda è descritta da un elemento di tipo `queue`.

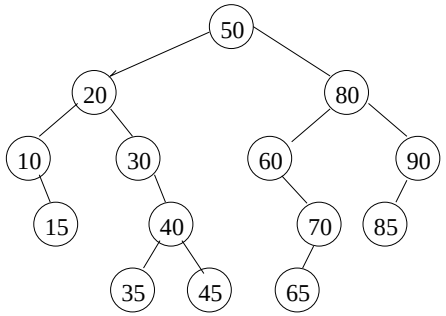
- `queue* newQueue()`
  Crea una coda vuota e restituisce il suo indirizzo.
- `void queueIn(int n, queue *q)`
  Inserisce l'elemento `n` nella coda `q`.
  Il valore di `q` non è modificato.
- `void queueOut(queue *q)`
  Toglie un elemento dalla coda `q`.
  Il valore di `q` non è modificato.
- `int queueFirst(queue *q)`
  Restituisce il valore del primo elemento della coda `q`.
- `int queueIsEmpty(queue *q)`
  Restituisce 1 se la coda `q` è vuota, 0 altrimenti.
- `void queueDestroy(queue *q)`
  Elimina la coda `q`.

Alcune operazioni potrebbero causare errore in esecuzione

Esempio: visita per livelli di un albero

Nella visita per livelli (*level order*) di un albero binario viene visitata prima la radice, poi i nodi di profondità (livello) 1, poi i nodi di profondità 2, ecc.

Nell'albero binario



l'ordine di visita dei nodi è:

50 20 80 10 30 60 90 15 40 70 85 35 45 65

A differenza delle altre funzioni di visita, la visita per livelli non ha una naturale formulazione ricorsiva.

La soluzione più semplice è di gestire l'ordine di visita mediante una coda in cui vengono inseriti gli *indirizzi* dei nodi da visitare.

Gli elementi della coda devono avere tipo `node*`.

Descrizione dell'algoritmo

Per visitare un albero di radice `r` non vuoto:

(1) Si crea una coda vuota Q.

(2) Si inserisce in Q la radice `r`

(3) Finché Q non è vuota:

  (3.1) Si estrae il primo nodo `p` di Q.

  (3.2) Si visita `p` e si mettono in Q, se definiti, il figlio sinistro e il figlio destro di `p`.

Verificare la correttezza dell'algoritmo sull'albero dell'esempio precedente.

La funzione `levelorder()` visita per livelli l'albero di radice `r` e stampa i nodi visitati.
Si assume che `r` non sia vuoto.

```
void levelorder(node *r){
    node *p;
    queue *q;
    q = newQueue();  // coda vuota
    queueIn(r, q);
    while (!queueIsEmpty(q)) {
            p = queueFirst(q);
            queueOut(q);
            printf("%d", p->key);
            if(p->left!= NULL)
                queueIn(p->left, q);
            if(p->right!= NULL)
                queueIn(p->right, q);
        }
    queueDestroy(q);
}
```

Possibili implementazioni

(1) Array allocato dinamicamente.

La gestione non è però semplice come nel caso dello stack (vedere il libro).

(2) Lista.

Occorre mantenere l'indirizzo al primo e all'ultimo elemento della coda.

In entrambe le soluzioni tutte le operazioni hanno complessità $O(1)$.

Coda di `int` implementata mediante una lista

Il tipo `queue` è definito come una struttura `struct queue` che contiene i campi `first` (primo elemento della coda) e `last` (ultimo elemento della coda).

Un elemento della lista che rappresenta gli elementi in coda è rappresentato dalla struttura `struct queueElement`

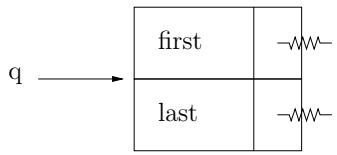
```
struct queueElement {  // elemento della lista
    int info;
    struct queueElement *next;
};

typedef struct queueElement  queueElement;

struct queue{
    queueElement *first;
    queueElement *last;
};

typedef struct queue  queue;
```


Creazione coda vuota

La funzione `newQueue()` crea una nuova coda vuota e restituisce il suo indirizzo.

La coda vuota è rappresenta da un elemento di tipo queue in cui i campi `first` e `last` valgono `NULL`.

```
queue* newQueue(){
 queue *q;
 q = malloc(sizeof(queue));
 q->first = q->last = NULL;
 return q;
}
```

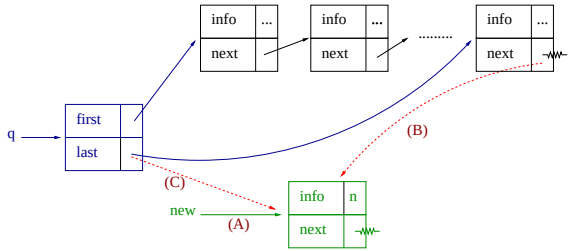


Inserimento

La funzione `queueIn()` inserisce l'elemento `n` nella coda descritta da `q`. Il valore di `q` non cambia.

Occorre creare un nuovo elemento `new` di tipo `queueElement`.

- Se la coda è vuota, l'elemento `new` diventa l'unico elemento della lista `q->first`.
- Se la coda non è vuota, l'elemento `new` viene inserito nella lista dopo `q->last`.



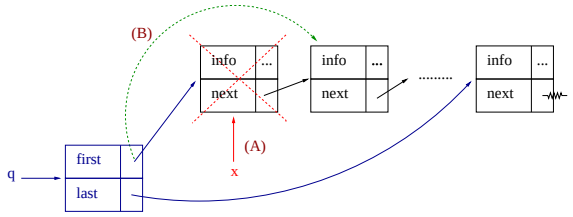

```
void queueIn(int n, queue *q){
  queueElement *new;
  new  = malloc(sizeof(queueElement)); // (A)
  new->info = n;
  new->next = NULL;  // new e' l'ultimo elemento della lista
  if( queueIsEmpty(q) ) // la coda e' vuota
    q->first = q->last = new;
  else{ // la coda non  e' vuota
    q->last->next = new; // (B)
    q->last = new;       // (C)
  }
}
```


Eliminazione di un elemento

La funzione `queueOut()` elimina il primo elemento della coda q, cioè l'elemento `q->first`. Il valore di q non cambia.

Si assume che la coda q non sia vuota.

- Se la coda contiene un solo elemento, `q->first` e `q->last` devono assumere il valore `NULL`.
- Altrimenti, il nuovo valore di `q->first` è l'indirizzo del secondo elemento della lista `q->first` di partenza (e `q->last` non cambia).




```
void queueOut(queue *q){
  queueElement *x;
  if(q->first == q-> last){ // la coda ha un solo elemento
      free(q->first);
      q->first = q->last = NULL;
  }
  else{  // la coda ha piu' di un elemento
    x = q->first;    // (A):  x e' l'elemento da cancellare
    q->first = x->next;  // (B)
    free(x);
  }
}
```


Valore primo elemento

La funzione `queueFirst()` restituisce il valore del primo elemento della coda descritta da q. Si assume che q non sia vuota.

```
int queueFirst(queue *q){
    return q->first->info;
}
```

Verifica coda vuota

La funzione `queueIsEmpty()` restituisce 1 se la coda descritta da q è vuota, 0 altrimenti.

```
int queueIsEmpty(queue *q){
    return q->first == NULL;
}
```

Eliminazione della coda

Per cancellare la coda descritta da q occorre prima cancellare gli elementi della lista `q->first`, quindi la struttura q.

```
void queueDestroy(queue *q){
  queueElement *x, *p;
  x  = q->first;
  while(x != NULL){
    p = x->next;
    free(x);
    x = p;
  }
  free(q);  // cancella la coda
}
```


Accesso a file esterno

Per poter accedere a un file, occorre aprirlo con la funzione `fopen()` il cui prototipo è:

```
FILE *fopen(char *name, char *mode)
```

dove:

- `FILE` è un tipo definito in `<stdio.h>`.
- `name` è il *nome* del file esterno da aprire.
- `mode` specifica la *modalità* con cui il file deve essere aperto.

Il valore restituito è un puntatore a un oggetto di tipo `FILE*` (*file pointer*) oppure `NULL` in caso di errore.

Un oggetto di tipo `FILE` è una struttura contenente le informazioni relative al file necesssarie per le operazioni di lettura e scrittura (indirizzo di un buffer, posizione corrente nel buffer, modalità con cui il file è stato aperto, se è stata raggiunta la fine del file, ecc.).

Esempi di modalità di apertura di un file

- `"r"` (*lettura*).

  Dà errore (ossia, restituisce `NULL`) se il file non esiste.
- `"w"` (*scrittura*).

  Se il file non esiste viene creato, se esiste si perdono i dati in esso contenuti.
- `"a"` (*append*).

  Se il file non esiste viene creato, se esiste i dati vengono scritti in fondo al file e non si perdono i dati in esso contenuti.
- `"rb"`, `"wb"`, `"ab"`.

  Analogo ai precedenti per file binari.

Per operare su un file si utilizza *esclusivamente* il puntatore restituito da `fopen()`. Un file può essere aperto in più di una modalità.

Al termine delle operazioni su un file, la connessione creata da `fopen()` va chiusa mediante una chiamata a `fclose()`, a cui va passato il descrittore del file da chiudere.

Funzioni di input/output

Le funzioni `getc()` e `putc()` sono analoghe a `getchar()` e `putchar()`, ma hanno un parametro in più che indica il file su cui va effettuata l'operazione.

```
int getc(FILE *fp);
int putc(int c, FILE *fp);
```

La funzione `getc()` restituisce il prossimo carattere del file associato a `fp` (descrittore di file aperto in lettura). Restituisce `EOF` in caso di errore o se si è raggiunta la fine del file.

La funzione `putc()` scrive il carattere `c` sul file associato a `fp` (descrittore di file aperto in scrittura). Restituisce il carattere scritto oppure `EOF` in caso di errore.

Input/output formattati

```
int  fscanf(FILE *fp, char *format, ...)
int  fprintf(FILE *fp, char *format, ...)
```

Sono identiche a `scanf()` e `printf()`, ma hanno come primo argomento il file pointer relativo al file su cui operare.

Quando un programma C viene eseguito, vengono aperti tre file e forniti i relativi puntatori (dichiarati in `stdio.h`) che hanno nome:

- `stdin` (*standard input*), per default associato alla tastiera.
- `stdout` (*standard output*), per default associato allo schermo.
- `stderr` (*standard error*), per default associato allo schermo.

Le istruzioni

```
c = getchar();         putchar('c');     printf("word");
printf("n= %d", n);        scanf("%d", &n );
```

sono equivalenti rispettivamente alle istruzioni

```
c = getc(stdin);   putc('c',stdout);   fprintf(stdout,"word");
fprintf(stdout,"n= %d", n);    fscanf(stdin, "%d", &n );
```


Come già visto, molti sistemi operativi permettono di eseguire un programma *redirigendo* standard input (generalmente con <) e standard output (generalmente con >); alcuni sistemi permettono anche la redirezione dello standard error (ad esempio con 2> o con >&).

Il programma esegue le operazioni di input/output riferendosi esclusivamente ai descrittori `stdin`, `stdout`, `stderr` e *non* è consapevole di eventuali redirezioni.

Esempio

Le seguenti linee di codice copiano carattere per carattere il contenuto di due file; i nomi dei file (al massimo `MAX` caratteri) sono inseriti dall'utente.

```
char word[MAX+1];
int c;
FILE *in, *out;
printf("Nome del file sorgente: ");
scanf("%s", word);
if ( (in =fopen(word,"r")) == NULL){
  ... IL FILE SORGENTE NON ESISTE ....
  }
printf("Nome del file destinazione: ");
scanf("%s", word);
out = fopen(word, "a");
while ((c=getc(in)) != EOF)   // c e' il prossimo carattere di in
    putc(c, out);  // c e' scritto in out
fclose(in);   // chiusura dei file
fclose(out);
```